– COVER PAGE —

Paper submitted to the 25th VLDB Conference.

Contact author: Wolfgang Hoschek
Contact mail: Wolfgang Hoschek, CERN IT Division, CH - 1211 Geneva 23, Switzerland.
Contact email: wolfgang.hoschek@cern.ch
Contact tel.: +41 22 76 78089

Reference number: **europe88** (Research paper)

Title: **Partial range searching in OLAP data warehouses**

Full author list:

Wolfgang Hoschek
CERN IT Division
CH - 1211 Geneva 23, Switzerland
wolfgang.hoschek@cern.ch

This page intentionally left blank.

Partial range searching in OLAP data warehouses

Wolfgang Hoschek*
CERN IT Division - European Laboratory for Particle Physics
1211 Geneva 23, Switzerland
wolfgang.hoschek@cern.ch

Abstract

We study physical database design for OLAP range queries. A useful DBMS should support the continous spectrum from low to high dimensional range queries with small to large selectivities operating on low to high cardinality attribute domains, a nontrivial problem. A set of heuristics is collected: multidimensional indexes are good for queries on most of the indexed dimensions, clustered files are good for queries on few dimensions, lossy compression can be useful for pruning, as well as other heuristics. We then suggest judiciously combining these observations with the state of the art in multidimensional indexes, bitmap indexes, lossily-compressed indexes, and others, to provide a design that supports a broad variety of range queries. Our technique is a hybrid composed of a variant of the extended pyramid tree, columnwise clustering, minmax based lossless and bin based lossy compression. This combination offers advantages over individual indexes, as the advantages of each structure can be multiplied.

1 Introduction

On-line analytic processing (OLAP) applications like scientific and commercial data exploration, knowledge discovery and data mining frequently use range queries to select objects of interest from massive high dimensional object sets stored in data warehouses. A number of properties distinguish OLAP object sets from object sets used in traditional On-line Transaction Processing (OLTP) applications. *First*, OLAP object sets can contain very large amounts of objects, easily in the range of 10^6 to 10^{10} objects. For example, the high energy physics experiment CMS [CMS96] at CERN will record objects derived from proton-proton collisions observed in a particle detector. An expected number of 10^9 objects per year will be accumulated over a lifetime of 15 years. *Second*, OLAP object sets are typically high dimensional. Their objects consist of dozens, hundreds or even thousands of attributes (dimensions). For example, in Y2K the COMPASS experiment at CERN will produce an object set on the order of 200 Terabytes [Col96]. Already today the experiment NA48 at CERN is recording data at a sustained rate of 20 MB/sec [PS98]. Support for efficient analysis of massive materialized views is to be provided. *Third*, such object sets are read-mostly, either never being updated or being updated only infrequently in large batch jobs during which the database may be off-line. Adding new objects is often also done in large batches simply constructing a new partition, filling it and logically (but not physically) attaching it to the database. This way, a database neither need to be taken off-line nor indexes reorganized. *Fourth*, OLAP object sets have a long lifetime. They are often kept and queried for months or years.

Recall, that OLAP and OLTP object sets also share common properties. In many cases, distinct objects are considered to be entirely uncorrelated, i.e. statistically independent from each other, resulting in object order to be irrelevant from the user's point of view. Attributes of a single object can well be correlated. Values of an attribute are often not uniformly distributed but highly skewed, i.e. most attribute values populate dense areas. For example, the attributes `age` and `salary` are usually correlated, with elder employees often earning more than young ones. `Salary` is also strongly skewed, since few people earn more than $100K a year, but most people's income fits within a

*Also affiliated with Dep. of Information Systems, Institute of Applied Computer Science, University Linz, Austria

Submitted to the 25th VLDB Conference
Edinburgh - Scotland - UK - 1999

certain realistic range.

OLAP range queries as well exhibit specific properties that distinguish them from OLTP range queries. Data exploration is an iterative process during which users frequently issue range queries, analyze their results to improve understanding e.g. of the nature of a subatomic Higgs particle, which in turn leads to improved selection criteria used in a subsequent query-analysis step. *First*, although at any given moment in time during production most frequently used query patterns of the past can be determined, future query patterns are to a significant degree unpredictable at index definition and reorganization time. Shape, location, and selectivity of range queries can strongly vary because a given object set can be used by many users for many different analysis use cases which moreover can significantly change over time with shifting interests. Further, access patterns are made less predictable by the fact that a user can be member of one of dozens of institutions loosely coupled in large global collaborations, as is, for example, the case in high energy physics experiments, earth and climate studies and in astronomy. For example, range queries can be exact range queries, i.e. operating on all object attributes. They can as well be partial range queries, i.e. operating on a possibly small subset of attributes. In many cases, partial range queries in a few attributes are more common than exact range queries. Queries can cover large or small volumes of domain space. *Second*, queries in OLAP applications do not necessarily follow the data distribution of an object set. They can operate on densely populated areas of data space (clusters) as well as on almost empty data space. Therefore, the volume of domain space covered by a range query is not necessarily correlated with its result set size. For example, a CMS application searching for the yet undetected Higgs particle will be confronted with a set of 10^9 objects which is expected to contain only of the order of 10^2 Higgs particles, if the particle exists at all. The properties of a Higgs particle are not fully understood. In this context it is unknown whether Higgs particles "live" in data space within dense clouds of noise objects or within virtually empty space. As a consequence, queries covering small volumes can have large result set size whereas large volumes can yield small result set size.

It follows, that determining how to index which attributes to obtain optimal performance for an inhomogeneous query mix is a nontrivial task. However, a useful DBMS must not only efficiently support certain types of range queries but cover the whole continous spectrum from low to high dimensional range queries with small to large selectivities operating on low to high cardinality attribute domains. In the OLAP arena, access methods researchers are facing challenges and opportunities different from OLTP. On the one hand side, unpredictable access patterns are increasingly becoming a major problem. On the other hand side, the read-mostly nature and long life time of OLAP object sets opens new opportunities to build integrated pipelined indexes relying on combinations of multidimensional indexes, clustered files, lossy and lossless compression, accurate histograms as well as other mechanisms. Such combinations offer advantages over individual indexes, as the advantages of each structure can be multiplied.

A considerable amount of work has been done in attempting to solve the problem of indexing high dimensional space, both for exact range queries and nearest neighbor search. Most recent offsprings of research activity in the field include the (extended) Pyramid tree [BKB98], VA-file [WSB98], BV-tree [Fre95], UB-tree [Bay97], X-tree [BKK96] and TV-tree [LJF94]. For surveys of the field see [BK98], [GG96]. Variants of the B-tree [Com79] are popular for one-dimensional indexes. A widespread index structure not specifically tied to low or high dimensionality is the bitmap with its myriads of variants using range-encoding, bit slicing, partitioning, compression or combinations thereof [CI98], [WB98], [OQ97] and [WY96]. [Sch94] discusses parallel range partitions. The VA-file [WSB98] uses bin based approximations in an intermediate layer for nearest neighbor search. For an overview of data warehousing see [BDF+97], [CD97a], [CD97b] and [Gup97]. A survey on query optimization in relational databases is given in [Cha98].

In this paper, we reshape some of these indexes to make them combinable. We also introduce new concepts were need arises. The new structures support a broad variety of range queries. Furthermore, they improve on their ancestors in several ways. Whereas the (extended) pyramid tree is limited to queries close to exact range queries, the foreign pyramid tree can effectively be used for exact and partial range queries without duplicating data objects. For low and medium cardinality attribute domains, the minmax index shows strong speedups over a plain columnwise clustered index. Next, the approximate index attacks the problems caused by high cardinality attribute domains. It is a framework modelling the constraints under which any lossily compressed index operates when obtaining both probabilistic and exact answers, regardless of the underlying implementation. Last, a concrete implementation of an approximate index is proposed, the quantile index. For probabilistic search it shows strong speedup rates over a columnwise clustered index. For exact seach it can often but not always contribute moderate to large speedups.

This paper is organized as follows. Section 2 clarifies essential terminology. Next, Section 3 motivates and describes our overall approach. Sections 4, 5 and 6 are the main contributions of this paper. Section 4 proposes the foreign pyramid tree, a variant of the extended pyramid tree. Section 5 analyzes properties of

the well known columnwise clustered index and proposes a compact variant termed minmax index. Next, Section 6 proposes the abstract approximate index and a concrete implementation of it, the quantile index. Finally, Section 7 concludes and refers to ongoing work.

2 Terminology

An object type is defined by its d attributes and their associated domains. In order to be meaningful for a range query, upon each domain an order relation must be defined. To simplify description and later analysis, we adopt the assumption of [BKB98] restricting a domain to be the floating point unit domain $[0,1]$. Note, that we can impose this assumption without loss of generality, since it is always possible to define an on-the-fly transformation from a given domain to the unit domain. Therefore, an object can be viewed as a d-dimensional point located in the unit hypercube $[0,1]^d$. Throughout this paper we will use the terms *attribute* and *dimension* interchangeably. An object set $S = \{O_0, .., O_{N-1}\}$ is given by a set of N such points. A *range query* Q in k of d dimensions is equivalent to a k-dimensional hyperrectangle and given by

$$Q = ([qmin_0, qmax_0], ..., [qmin_{k-1}, qmax_{k-1}])$$

with $qmin_i, qmax_i \in [0,1], qmin_i \leq qmax_i, 1 \leq k \leq d$.

"*Find all objects with* $x \in [0.2, 0.5]$ *and* $y \in [0.1, 0.9]$ *and* $z \in [0.2, 0.3]$" is an example for a range query. A *partial range query* is a range query with less than all attributes specified, i.e. with $1 \leq k < d$. An *exact range query* is a range query with all attributes specified, i.e. with $k = d$. A partial range query can further be seen as an exact range query with all unspecified attributes represented by the full extension of the unit domain $[0,1]$. The *result set* of an exact range query is the set of all objects or object identifiers satisfying the query. The *spatial selectivity* ss of a range query is the fraction of domain space covered by it and given by $ss = \prod_{i=0}^{k-1} qmax_i - qmin_i$. The *result selectivity* rs of a range query is the fraction of the object set selected by it and given by $rs = \frac{resultSetSize}{sourceSetSize}$. In the case of uniformly distributed attribute values, there holds $ss = rs$. If subsequently we use the unqualified term *selectivity*, we refer to a result selectivity.

3 MARS

Although a considerable amount of work has been done to cope with each of the properties of OLAP object sets and range queries separately, to the authors best knowledge, no single index data structure can adequately handle all properties at the same time. *First*, B-trees, bitmaps and their relatives by definition cannot adequately prune multidimensional space. *Second*, intuitively it is clear, that any multidimensional index data structure partitioning the space can only poorly perform if confronted with a partial range query, since the l partitions of $d-k$ indexed dimensions do not contribute at all to pruning. Therefore, the pruning ability of a multidimensional index degrades with decreasing k. In addition, the l partitions of $d-k$ indexed dimensions can unnecessarily fragment an object set, which often increases random access at the cost of linear access. This is particularly unfortunate in the case of highly selective queries. Other index techniques, too, perform well for certain use cases, but fail for others.

This suggests, that a single index structure alone is unlikely to be able to provide adequate efficiency for OLAP requirements and that rather an integrated combination of several index structures is most likely to provide the greatest potential. We think that it is important to be cognizant of the combined power of using multiple indexing techniques and how that interacts with query optimization. Motivated by this insight, we have been looking for components which

- to a large degree complement each others disadvantages
- can effectively be combined in a filter pipeline to contribute multiplicative pruning
- can be plugged together seamlessly, yet are independent enough to allow for different query execution plans for different kinds of range queries
- are simple yet effective

There are good reasons why *simplicity* is a key requirement for useful index structures. For reliable global cost models, it is important to build upon simple indexes. Complex indexes are often (if at all) accompanied by complex and/or fragile cost models which are hard to combine in a query optimizer. This is part of the reasons why many advanced index structures do not find their way into commercial DBMS's. Even if some of them would be incorporated, the majority of DBA's (who are not access method researchers) would simply find them incomprehensible, offering little practical guidance as to how and when they should be used and when alternatives are preferable.

Three simple and effective index structures were designed, the *foreign pyramid tree*, the *minmax index* and the *quantile index*, together forming *MARS* (Multi Attribute Range Searching). For example, the fewer attributes a range query specifies, the worse a foreign pyramid tree performs, whereas both minmax index and quantile index perform the better. The reverse also holds. As a further example, the more selective a range query, the less a foreign pyramid tree is able to prune the search space, whereas both other index structures can considerably reduce the amount of I/O

needed to be performed. Not only do these components to a large degree complement each others disadvantages, but, queried in the right order, they can mutually benefit from each other. For example, a foreign pyramid tree can often strongly prune the search space for a range query close to an exact range query, while both other index structures can contribute further speedup factors during subsequent query processing steps. A query optimizer has foreign pyramid tree, minmax index and quantile index at its disposal. It may, for example, decide to visit these indexes in a pipeline establishing order, visiting a foreign pyramid tree first, then proceeding to query a quantile index and finally query a minmax index. Before we describe these three index data structures which are largely independent from each other, let us briefly report how supporting statistics are obtained and used.

3.1 Collecting statistics

Given the read-mostly nature and long lifetime of OLAP object sets, spending time to gather accurate statistics describing an object set is affordable. Statistics are collected before bulk loading and never need to be updated again. The technique reported in [MRL98] is applied to compute approximate equi-depth histograms (one-dimensional quantiles, including medians) representing an entire object set. The technique requires only one pass over the object set and limited memory while giving explicit and customizable approximation guarantees. Histograms will later on be used for selectivity estimation, query optimization and lossy compression. To later on be able to compress attribute values without loss, for each attribute minimum and maximum of its values are computed in the same pass.

4 Multidimensional index

The extended pyramid tree [BKB98] was chosen as basis for a multidimensional index data structure for a number of reasons. It is simple and effective. Bulk loading a pyramid tree is fast, which is important to allow to rebuild indexes upon changing access patterns. For exact range queries the extended pyramid tree scales well in space and time with the number of dimensions and the number of stored data objects. The pyramid tree processes range queries the better the higher dimensional space and query are. The closer a partial range query matches an exact range query, the faster queries are processed. For partial range queries with decreasing dimensionality, its performance gracefully (near linearly) degrades to a linear scan.

Figure 1 shows the pruning behavior of a pyramid tree for range queries with varying dimensionality and spatial selectivity. Index dimensionality was $d = 10$. Data was uniformly distributed. For each plot point 25 randomly located hypercube shaped range queries contained in the unit hypercube were computed. An affected volume of 0.2 means that 20% of all objects had to be examined. The figure shows the importance of exact range queries for a pyramid tree. Where $k \leq d$, pruning ability degrades with decreasing k near linearly to a linear scan. Where $k > d$, query dimensionality was shrunk to d dimensions by omitting $k - d$ dimensions, thereby increasing selectivity. In these cases, it can be seen that if an efficient way can be found to externally index the remaining $k - d$ dimensions, performance can often be improved over a single index with dimensionality$> k$.

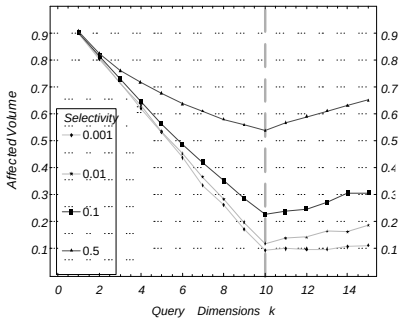

Figure 1: Pruning behaviour of pyramid tree

Let us briefly recall the ideas of the extended pyramid tree. It is an index data structure particularly designed for range queries in high dimensional space. It views the full search space as a unit hypercube $[0, 1]^d$, where d is the number of attributes (dimensions) indexed. A hypercube of dimensionality d has $2 \times d$ "surfaces", each "surface" being $(d - 1)$ dimensional. The full search space is partitioned into $2 \times d$ hyperpyramids, where each pyramid has as its base one of the $2 \times d$ "surfaces" and as its top the center point of data (an approximation of the d-dimensional median), as shown in figure 2a). The technique defines a geometric transformation mapping d-dimensional points (objects) to one-dimensional points called *pyramid values*. Each pyramid value consists of the *pyramid* an object falls into as well as the *height* of the object within its pyramid, as shown in figure 2b). A concatenation trick is used to merge both components into a one-dimensional pyramid value. The pyramid tree physically organizes its points with the help of a one-dimensional index data structure like peels of an onion, e.g. by using a B^+-tree. Attribute values of a single object are clustered together. Range queries are processed by determining the parts of pyramids affected by a query as shown in figure 2c). Each affected part contains contiguously stored points. Those points are read in and checked against the query to determine whether they really qualify.

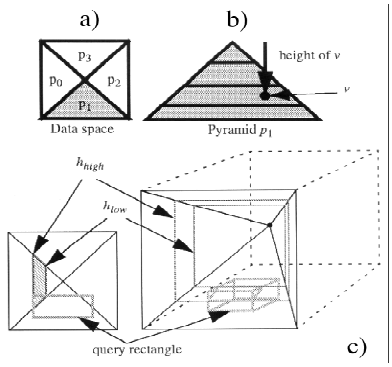

Figure 2: Pyramid tree [BKB98]

4.1 Foreign pyramid tree

We adopt the main ideas of the extended pyramid tree. However, it is modified in two ways to improve bulk loading, and more importantly, to allow clustering methods suited for partial range searching. Recall that an (extended) pyramid tree clusters attribute values of a data object together. Although optimal for exact range queries, this means that for partial range queries potentially many irrelevant attributes need to be read during query processing. We therefore modularize the closed extended pyramid tree such that clustering decisions are delegated to an external index. Now the extended pyramid tree can be combined with other index structures (e.g. a columnwise clustered index) in many interesting ways. The new index structure is a very fast prefilter layered on top of another arbitrary index structure holding data objects.

We call the new structure *foreign pyramid tree* to stress its most important property, which lies in not storing data objects but merely a key pair list in a B^+-tree, i.e. a list of (key, pyramid value) pairs physically sorted and logically indexed by ascending pyramid value. Each key pair represents a data object. Keys (integers) are assigned to data objects by their order. Data objects are stored in an arbitrary external index (preferably sorted by key) which implements clustering methods appropriate for partial range searching. If the external index is sorted by key, then a key can be seen as a row id.

Definition 1 (Key) *Let $S = \{O_0, .., O_{N-1}\}$ be a set of N objects, pv_{obj} be the pyramid value of a particular object, the list SS be S sorted ascending by pyramid value, i.e. such that $\forall_{0 \le i < N-1} : pv_{SS_i} \le pv_{SS_{i+1}}$, then the key of an object obj is defined as $key_{obj} = (i | SS_i = obj)$.*

Definition 2 (Key pair and key pair list) *The key pair KP of an object obj is defined as $KP_{obj} = (key_{obj}, pv_{obj})$ and the key pair list KL of S is defined as $KL = (KP_{SS_0}, ..., KP_{SS_{N-1}})$.*

Figure 3: Affected key intervals

Query processing takes as input a range query and produces as output a list of affected $[minkey, maxkey]$ intervals representing qualificant objects which need to be checked against the query. In our implementation these intervals are identical to $[minRowId, maxRowId]$ intervals of an external columnwise clustered index sorted by key. The output list is computed as follows. First, the parts AP of pyramids affected by a query are computed without any I/O exactly as done by the extended pyramid tree: A list AP is set to be empty. Then, for each pyramid it is determined if it is affected by the query. If so, the minimal pyramid value interval $[minpv, maxpv]$ containing the query is computed and appended to the list AP. For details, see [BKB98].

Proposition 1 (Affected parts)
Let pv.pyramid and pv.height denote the pyramid and height of a pyramid value pv, respectively. Then affected parts AP of an (arbitrary) pyramid tree with dimensionality d take the shape $AP = ([minpv_0, maxpv_0], ..., [minpv_t, maxpv_t])$, with $0 \le t < 2 \times d, \forall_{0 \le u \le t} : (0 \le minpv_u.pyramid = maxpv_u.pyramid < 2 \times d, 0 \le minpv_u.height \le maxpv_u.height \le 0.5), \forall_{0 \le u < t} : maxpv_u < minpv_{u+1}$.

In a second step, affected key intervals corresponding to affected parts are determined according to the following definition.

Definition 3 (Affected key intervals) *Let $KP.pv$ denote the pyramid value of a key pair KP, then the affected key intervals AK of affected parts AP are defined as $AK = ([minkey_0, maxkey_0], ..., [minkey_t, maxkey_t]), \forall_{0 \le u \le t} : (KL_{minkey_u}.pv \ge minpv_u, (\forall_{0 \le i < minkey_u} : KL_i.pv < minpv_u), KL_{maxkey_u}.pv \le maxpv_u, \forall_{maxkey_u < j \le t} : KL_j.pv > maxpv_u)$.*

In other words, the $minkey$ of a $minpv$ corresponds to the position of the leftmost $pyramidValue \ge minpv$ in a key pair list. The $maxkey$ of a $maxpv$ corresponds to the position of the rightmost $pyramidValue \le maxpv$ in a key pair list. Figure 3 depicts these relationships. Affected key intervals can be found by traversing the foreign pyramid tree's B^+-tree in a straightforward and efficient manner. Assuming that internal nodes of a B^+-tree can be kept in main memory cache, looking up affected key intervals requires at

most $2 \times (t+1)$ page reads $(0 \leq t < 2 \times d)$. For example, in a 10-dimensional foreign pyramid tree at most 40 pages need to be read. The average case is close to the worst case. $\sum_{i=0}^{t} 1 + |AK_i|$ objects of the external index need to be examined further.

To summarize, because the foreign pyramid tree delegates clustering decisions, it can effectively be used for exact and partial range queries without duplicating data objects, whereas the (extended) pyramid tree is limited to queries close to exact range queries.

We now improve on the bulk loading mechanism known so far. Bulk loading an extended pyramid tree requires knowledge about the d-dimensional median of an object set. [BKB98] proposed a method to load an extended pyramid tree while computing an approximation of the d-dimensional median on the fly. Their method requires the tree to be rebuilt whenever the constantly adjusted median has moved more than a given threshold, taking at most a logarithmic number of rebuilds. In contrast, a foreign pyramid tree relies on medians representing the entire object set to be provided by the external statistics collecting component already described. Therefore, a foreign pyramid tree need not be rebuilt during bulk loading. By the time a foreign pyramid tree is to be loaded, medians are already computed, resulting in superior load performance.

5 Columnwise clustered index

A *columnwise clustered index* is a relation with columnwise clustering. It is, for example, the default in Sybase IQ [Syb97]. It will serve as the root for several more refined index structures described later on. The main idea behind this index is to avoid having to read in attributes which are irrelevant to a partial range query. For example, a partial range query specifying only five out of hundred attributes shall not waste I/O in reading 95% irrelevant attributes. In contrast to a (foreign) pyramid tree, the closer a range query matches an exact range query, the less pruning a columnwise clustered index can contribute. Or, the other way round, the closer a range query matches a one-dimensional range query, the more pruning it can contribute. The columnwise clustered index is not only simple but also robust under unpredictable access patterns, a fundamental advantage in OLAP.

Query execution takes as input a column, a query range $R = [qmin, qmax]$ and a *checklist*, i.e. a sorted list of row ids to be checked. It returns as result a modified checklist containing row ids of attribute values contained in the query range (*members*). The resulting checklist is used as input to query the next column, and so on. A checklist is modelled as a bit vector. To improve pruning efficiency a query estimator judges the selectivity of a range query for each column based on equi-depth histograms built before bulk loading. Columns are queried in order of ascending selectivity.

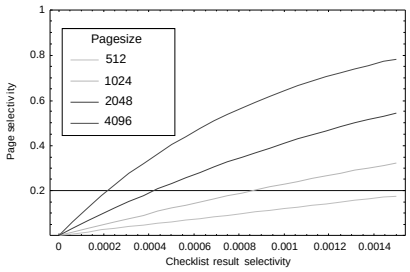

Figure 4: Estimated average page selectivity

The index supports two retrieval plans, *partial scan* and *full scan*. *Partial scan* reads only pages of a column containing attribute values that need to be tested. It reads attribute values referred to by the input checklist from "top to bottom", while skipping gaps between pages referred to by the checklist. For such direct access to be possible, a column holds a simple $O(1)$ lookup table which allows for a given row id to obtain its value by returning a reference to its page and the position within it. The retrieval plan *full scan* has the same interface as *partial scan*, but sweeps over a column without skipping page gaps.

5.1 Analysis

From a checklist, its page selectivity can be computed. Using *partial scan*, no speedup over *full scan* can be seen in typical disk-operating system pairs above a threshold of some $15 - 20\%$ page selectivity because these parts strongly favour linear access over random access. Below the threshold, decreasing page selectivity causes exponentially growing speedup over *full scan* [Hol98], [Hol97]. A columnwise clustered index uses the threshold as well as an estimate of the page selectivity an input checklist causes to a column to decide which retrieval plan to apply to a particular column.

Lemma 2 (Average page selectivity estimate) *Let N be the cardinality of a column, as be the bytes an attribute value of the column takes, s_{check} be the result selectivity of a checklist, pagesize be a disk partition's or DBMS page size [bytes], then, in the spirit of [Mal98], an estimate of the average page selectivity of the checklist is given by* $s_{page} = 1 - \frac{\binom{n \times (p-1)}{N \times s_{check}}}{\binom{n \times p}{N \times s_{check}}}$, *with* $n = \left\lfloor \frac{pagesize}{as} \right\rfloor, p = \left\lceil \frac{N}{n} \right\rceil, \binom{x}{y} = 1$ *if* $y > x$.

Figure 4 depicts estimates for varying checklist result selectivities and page sizes with $as = 4$. As can be seen, only checklists with very low result selectivity translate to a page selectivity below a threshold of some $15 - 20\%$. For example, for $pagesize = 4096$ bytes (512), only checklists with result selectivity $<$ 0.0002 (0.002) are worth to be executed with the retrieval plan *partial scan*. Note, that such low result se-

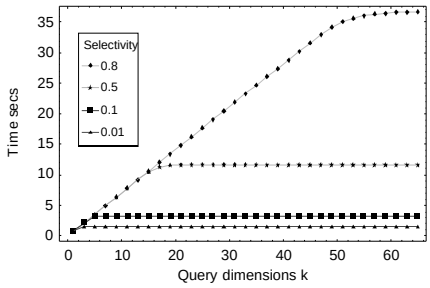

Figure 5: Est. read time in columnwise clust. index

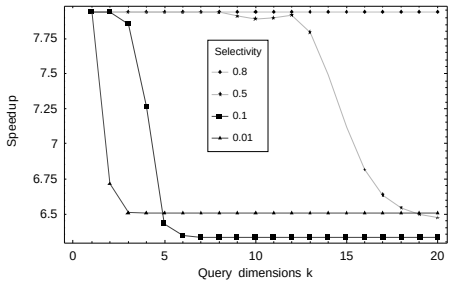

Figure 6: Estimated speedup of minmax index over columnwise clustered index

lectivities can often occur in high dimensional queries since selectivities are multiplicative.

It is surprising that many access methods use the number of page reads as unit for cost models. This unit often poorly reflects the actual I/O time, since an access method A reading 5 times more pages than method B can be just as fast in terms of I/O. (Cache hits play a minor role in OLAP, because queries usually retrieve massive amounts of data so that cache hits are rare in any case.) Here a cost model consisting of two parameters is used to estimate the time needed to execute a range query. First, the *readrate* for *full scan* over a single column at 100% page selectivity is measured. Second, the *speedup* of *partial scan* over *full scan* is measured as a function of page selectivity. Both parameters vary from one disk-operation system pair to another. Throughout this paper, we use the measurements of [Hol98], [Hol97] as basis for our examples, resulting in $readrate = 691$ pages/sec ($\triangleq 5.4MB/sec$) at $pagesize = 8192$ and a speedup curve growing exponentially with decreasing page selectivity below a threshold of 20% page selectivity.

The time needed to read a column with pg pages at page selectivity s_{page} is given by $t = \frac{pg}{readrate \times Max(1, speedup(s_{page}))}$. Figure 5 shows estimated read times in a columnwise clustered index for a k-dimensional range query with the same selectivity in each attribute. Parameters were $N = 10^6$, $as = 4$ (integer attributes), $readrate = 691$ $pages/sec, pagesize = 8192$.

5.2 Minmax index

In this subsection we use lossless compression on attribute values of low and medium cardinality domains. For query execution it is important to have a compression technique that, given a row id, allows to look up the associated attribute value in $O(1)$. A compression technique without this property could be useful, but is likely to be problematic because it must carefully balance CPU time against I/O time. We propose the *minmax index*, a small yet often effective extension of a columnwise clustered index. It can also be seen as a type of compressed bitmapped index. The minmax index carries the $O(1)$ property. In fact, it imposes almost no CPU overhead, resulting in high query execution performance even though it sometimes does not yield strong compression.

Before bulk loading a column, minimum and maximum of its attribute values are taken from the statistics collecting component and only the necessary constant number of bits are used to store a compact representation of an exact attribute value. Given an attribute normally taking *bits* bits, the compression coefficient cc is given by $cc = \frac{compressed}{uncompressed} = \frac{\lceil \log_2 1+max-min \rceil}{bits}$.

Figure 6 depicts estimated speedup rates of a minmax index over a plain columnwise clustered index. Parameters were $N = 10^6, pagesize = 4096$. Attribute size was $as = 4$ (integer attributes) in the columnwise clustered index. Compression coefficient was $cc = \frac{1}{8}$ in the minmax index. As can be seen, speedup is not constant equal to $\frac{1}{cc}$ since the lower a compression coefficient, the more values fit onto a page which in turn can increase page selectivity.

Although methods exist to compress floating point values without loss, the minmax index does not support one. In addition, if attribute values cover large parts of a high cardinality domain, its encoding mechanism cannot contribute much compression. The subsequently discussed indexes complement these disadvantages.

6 Approximate index

In this section, we attack the problems caused by high cardinality attribute domains. This work is motivated by the fact that in scientific data warehouses floating point attributes are very common. Unfortunately, such attributes are not well compressible and quickly decompressible without loss. The only applicable class of methods seems to be lossy compression. We propose the *approximate index*, an abstract index modelling the constraints under which any lossily compressed index operates when obtaining both probabilistic and exact answers. It is equipped with query processing techniques generally applicable regardless of the un-

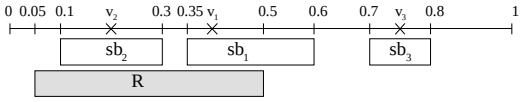

Figure 7: Members and Candidates

derlying implementation.

The approximate index is about trading accuracy for space. The idea is to store an approximation of each attribute value which takes an order of magnitude less space than the exact value, yet still allows to considerably prune the search space. Query execution can then operate on smaller objects, therefore needing an order of magnitude less I/O. The index can be used as fast stand-alone index, if only probabilistic answers to a query are needed. If, as in most cases, exact answers to a query are mandatory, the index can be used as a safe coarse grained fast prefilter layered on top of another index holding exact attribute values, e.g. a minmax index. An arbitrary lossy compression technique can be applied to approximate attribute values (e.g. bin based approximation, the fast wavelet transform, etc.).

For the correctness of query execution, an arbitrary lossy compression technique must store (relative or absolute) bounds on the approximation error it causes to a column, page or value. This is necessary for query execution to be able to reconstruct safe bounds on an approximate value, i.e. an interval which is guaranteed to contain the exact value. Of course, it is desirable to have safe bounds to be as tight as possible, reconstructed from as little space as possible. It is up to a concrete implementation to define how to compute and store approximations and how to reconstruct safe bounds.

Definition 4 (Safe bound) *Let v be an exact value, x be a (decompressed) approximation of v, then $sb = [min, max]$ is a safe bound on $x \Leftrightarrow v \in sb$.*

Let $R = [qmin, qmax]$ be a query range. Then it can be determined based on an approximate value alone, whether $v \in R$ or v *perhaps* $\in R$, i.e. whether x is a *member* or a *candidate* of R as follows.

Definition 5 (Member, Candidate, Qualificant) *x is a member of $R \Leftrightarrow sb \in R$. x is a candidate of $R \Leftrightarrow sb \notin R \wedge (sb.min \in R \vee sb.max \in R \vee R \in sb)$. x is a qualificant of $R \Leftrightarrow x$ is a member of $R \vee x$ is a candidate of R.*

Obviously, there holds x is a member of $R \Rightarrow v \in R$, as well as x is not a qualificant of $R \Rightarrow v \notin R$. Figure 7 depicts an example with $R = [0.05, 0.5]$. For $v_1 = 0.4$ we store a compact representation of, say $x_1 = 0.42$, and during query processing get $sb_1 = [0.35, 0.6]$ and x_1 is a candidate of R. For $v_2 = 0.2$ we store, say $x_2 = 0.22$, and get $sb_2 = [0.1, 0.3]$ and x_2 is a member of R. For $v_3 = 0.75$ we store, say $x_3 = 0.71$, and get $sb_3 = [0.7, 0.8]$ and x_3 is neither a member nor a candidate of R.

We distinguish two query use cases: a) probabilistic answers are sufficient, b) exact answers are mandatory, leading to two query execution plans, *probabilistic search* and *exact search*. Both plans can use *partial scan* and *full scan* as retrieval plans. Since the usage of the two retrieval plans is irrelevant to *probabilistic search* and *exact search* at the level described here, we subsequently will no more distinguish between them.

6.1 Probabilistic search

Probabilistic search is an execution plan for a use case that delivers truly probabilistic answers in the sense that a result set may contain false hits and miss true hits. The user specifies the degree of confidence his/her use case requires. He/she may e.g. decide to include in the result set only objects which have a probability of $\geq 95\%$ to satisfy a query. *Probabilistic search* takes as input a column, a query range $R = [qmin, qmax]$, a confidence threshold $ct \in [0,1]_{float}$ and a *probability checklist* $PCL = (PCL_o, ..., PCL_{N-1})$, with $PCL_i \in [0,1]_{float}$. Entries in the input probability checklist are expected to be set to 1 if the object associated with the entry should be checked, and 0 otherwise. A search returns as output a modified probability checklist containing the probabilities of objects with a probability to be contained in the query greater or equal to the confidence threshold. All other entries of the output are set to 0.

Assuming actual data distributions within safe bounds are unknown, it is reasonable to assume uniform distribution and uncorrelated attributes. In this limited environment, probabilistic answers are based on ratios of spatial volumes. We will later in Section 6.3 see that for concrete implementations data distributions are determined which allows to replace ratios of spatial volumes with precise probabilities. However, for now, we can view safe bounds on an approximated object as a range query.

Lemma 3 (Contained fraction) *Let A, B be range queries in the same k dimensions. Then the fraction f of B contained in A is given by $f_{A,B} = \prod_{i=0}^{k-1} \frac{|A_i \cap B_i|}{|B_i|}$.*

Now the probability that an exact object is contained in a given range query can be determined based on an approximated object alone according to the following lemma.

Lemma 4 (Object containment probability) *Let Q be a range query in k dimensions, $X = (x_0, ..., x_{d-1})$ be an approximation of a d-dimensional*

object O, then the probability P that O is contained in Q is given by $P = f_{Q,(sb_{x_0},...,sb_{x_{k-1}})}$.

Lemma 5 (Value containment probability) *Let $R = [qmin, qmax]$ be a query range, v be an exact value, x its approximation. Then the probability p that $v \in R$ is given by $p_{x,R} = \frac{|R \cap sb_x|}{|sb_x|}$.*

Range queries are executed as follows. Let $C = (x_0, ..., x_{N-1})$ be a column with approximate values. Let R be the query range for the column. For each i with $PCL_i > 0$ the probability checklist is modified as follows $PCL_i = \left\{ \begin{array}{cc} PCL_i \times p_{x_i,R} & \text{if } PCL_i \times p_{x_i,R} \geq ct \\ 0 & \text{else} \end{array} \right\}$. The obtained output probability checklist is used as input for a next column, and so on. As can be seen, objects below the confidence threshold are shortcut and no more need to be considered in subsequent columns.

6.2 Exact search

Exact search is an execution plan for a use case that delivers exact answers in the sense, that its output exactly distinguishes between three classes of objects: a) objects which are members, b) candidates, i.e. objects which might be members and c) objects which are neither. It does not depend on probabilities.

Exact search takes as input a column, a query range $R = [qmin, qmax]$ and a *qualificant checklist* (QCL), a bitvector with the same shape as used by a columnwise clustered index, i.e. $QCL = (QCL_o, ..., QCL_{N-1})$, $QCL_i \in \{true, false\}$. It returns as output a modified qualificant checklist containing *qualificants*, i.e. objects either being a member or a candidate. As additional output, *status checklists* (SCL) of the same shape containing members only are returned.

Definition 6 (Qualificant & status checklist) *Let $C = (x_0, ..., x_{N-1})$ be a column with approximate attribute values, then a qualificant checklist QCL is defined as $QCL = (QCL_0, ..., QCL_{N-1})$, with $QCL_i = \left\{ \begin{array}{cc} \texttt{true} & \textit{if } x_i \textit{ is a qualificant of } R \\ \texttt{false} & \textit{else} \end{array} \right\}$. A status checklist SCL is analogously defined as $SCL = (SCL_0, ..., SCL_{N-1})$, with $SCL_i = \left\{ \begin{array}{cc} \texttt{true} & \textit{if } x_i \textit{ is a member of } R \\ \texttt{false} & \textit{else} \end{array} \right\}$.*

Range queries are executed as follows. Let $C = (x_0, ..., x_{N-1})$ be a column with approximate values, R be a query range for a column. For each i the status checklist is set to $SCL_i = QCL_i$ `AND` x_i is a member of R. For each i the qualificant checklist is modified as follows: $QCL_i = QCL_i$ `AND` x_i is a qualificant of R. The output qualificant checklist is used as input for a next column, and so on. Status checklists are collected while successively sweeping over multiple

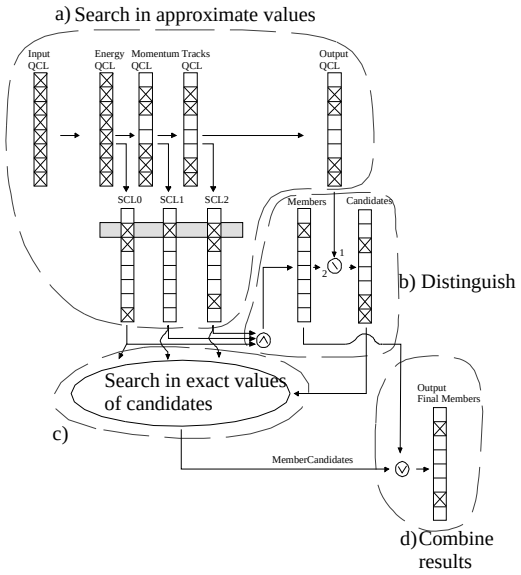

Figure 8: Exact search

columns. After all columns relevant for a range query have been processed, *exact search* returns to the query execution engine the last qualificant checklist and all status checklists. Figure 8a) depicts an example use case.

In a next step, candidates need to be checked further to exactly determine, whether they satisfy the query or not. This is done by looking up the exact values of candidates stored in an arbitrary external index, e.g. a minmax index. However, having applied the plan *exact search* the query execution engine now has status checklists at its disposal describing which attributes actually made an object a candidate. It therefore only needs to read and check those relevant attribute values.

The just computed qualificant checklist and all status checklists are given as input to a query execution plan operating on an external index holding exact values, as follows. First, the plan determines members. Members are objects which are contained in all status checklists at the same time. The memberlist ML can be found by intersecting statuslists (c.f. Fig. 8b). Next, candidates (CCL) are determined with a checklist difference operation, since candidates are objects which are contained in the last qualificant checklist, but not in the just obtained memberlist (c.f. Fig. 8b).

Let Q be a range query, $C_{j,i}$ be the exact value of position i in column j, $SCL_{j,i}$ be checklist entry i of status checklist j. Then exact values of candidates are searched as follows. For each column j, for each i in CCL the candidate checklist is modified according to: $CCL_i = CCL_i$ `AND` $(SCL_{j,i}$ `OR` $C_{j,i} \in Q_j)$. The shortcut order of boolean operators ensures that no more than the absolutely necessary exact values cause

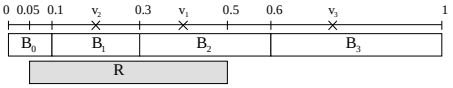

Figure 9: Bin Index

I/O. The loop eventually results in an output checklist CCL containing all candidates which turned out to be members (c.f. Fig. 8c). In a final step CCL is unionized with the memberlist ML and returned as result set of the query (c.f. Fig. 8d). Note that bitvector operations can be implemented extremely efficient, resulting in little CPU consumption for execution of both probabilistic and exact search.

6.3 Quantile index

Assume partitioning an attribute domain into disjoint subdomains (*bins*) such that the union of all subdomains is the entire domain. A *bin index* is a concrete implementation of an approximate index using a min-max index to store for each attribute value as approximation its bin value bv (an integer), i.e. the bin an attribute value falls into. At the price of accuracy, one can choose how much space to spend by deciding how much bins to allow, since a value from b bins takes $\lceil \log_2 b \rceil$ bits. For example, by using 16 bins one is able to approximate a 32 bit value (float or integer) with 4 bits.

Assume b bins $B = ([min_0, max_0), ..., [min_{b-1}, max_{b-1}])$ and a bin value bv of an exact value v such that $v \in B_{bv}, 0 \leq bv < b$. To fit bins into the general framework of the approximate index, we define the approximation x of a value to be its bin value and reconstructed safe bounds on the approximation to be the bin of a bin value, i.e. $x = bv, sb_x = B_{bv}$.

Figure 9 depicts an example with $B = ([0, 0.1), [0.1, 0.3), [0.3, 0.6), [0.6, 1])$ and $R = [0.05, 0.5]$. For $v_1 = 0.4$ we store a compact representation of $bv_1 = x_1 = 2$ and during query processing get $sb_1 = [0.3, 0.6)$ and x_1 is a candidate of R. For $v_2 = 0.2$ we store $bv_2 = x_2 = 1$ and get $sb_2 = [0.1, 0.3)$ and x_2 is a member of R. For $v_3 = 0.75$ we store $bv_3 = x_3 = 3$ and get $sb_3 = [0.6, 1]$ and x_3 is neither a member nor a candidate of R. Similar ideas are applied in the VA-file [WSB98], which uses bin based approximations in an intermediate layer for nearest neighbor search but differs from the bin index in that it clusters attribute values of an object together.

Since it is desirable to achieve good pruning in the average case, each bin should contain about the same number of values. Therefore bins are drawn from the equi-depth histograms (quantiles) already available from the statistics collecting component described in Section 3.1. The resulting index structure is called *quantile index*.

Probabilistic search.

Since the value distribution of attributes is known from histograms, the value containment probability from Section 6.1 can now be expressed precisely by $p_{x,R} = \frac{N(R \cap sb_x)}{N(sb_x)}$, where $N(r)$ is the number of values contained in range r, obtained from an equi-depth histogram, $sb_x = B_{bv}$ and $N(sb_x) = \frac{N}{b}$. The more bits stored and the more quantiles a histogram keeps, the higher the accuracy of this probability. We suggest histograms occupying one page with, say, 1000 quantiles.

In a bin index, probabilistic search can efficiently be implemented as follows. Precompute $p_{x,R}$ for all bins and temporarily store the probabilities in an array. While iterating over a column's approximate values, simply look up the probability to be contained in the query range in $O(1)$ in the array. Exact search uses similar precomputations not involving probabilities to determine whether an approximate value is a qualificant and a member. Thus, CPU comsumption for both plans is very little.

6.3.1 Analysis of exact search

Fig. 10 (a,b,c) depict estimated speedup rates of *exact search* with a quantile index over a plain columnwise clustered index. Number of bins and result selectivity are varied. Each plot point depicts one k-dimensional range query with the same given result selectivity in each dimension. Parameters were $N = 10^6, pagesize = 4096, as = 4$ (float attributes).

As expected, the smaller b is chosen, the larger the maximum speedup we can hope for gets. However, the smaller b, the higher dimensional a query needs to be to enjoy speedup. The larger b, the lower dimensional a query may be to enjoy some (moderate) speedup. The higher dimensional queries are, the larger speedup gets. However, for queries with large result selectivity a quantile index often cannot effectively prune the search space; too many candidates remain. To summarize, the overhead of querying an intermediate quantile index is often (but not always) small compared to the reduction of I/O it can contribute to the next query step. It is a good idea to keep more than one quantile index, each with a different number of bins (e.g. $4, 16, 256$).

Fig. 10 (d,e,f) show in greater detail how the speedup rates reported in Fig. 10 b) are assembled. They depict estimated read times for varying query dimensionality and result selectivities. Each plot point depicts one k-dimensional range query with the same given result selectivity in each dimension. Fig. 10 d) shows the time needed to execute queries in a plain columnwise clustered index. The usage of the execution plan *exact search* in a quantile index is depicted in Fig. 10 e). Read time requirements for looking

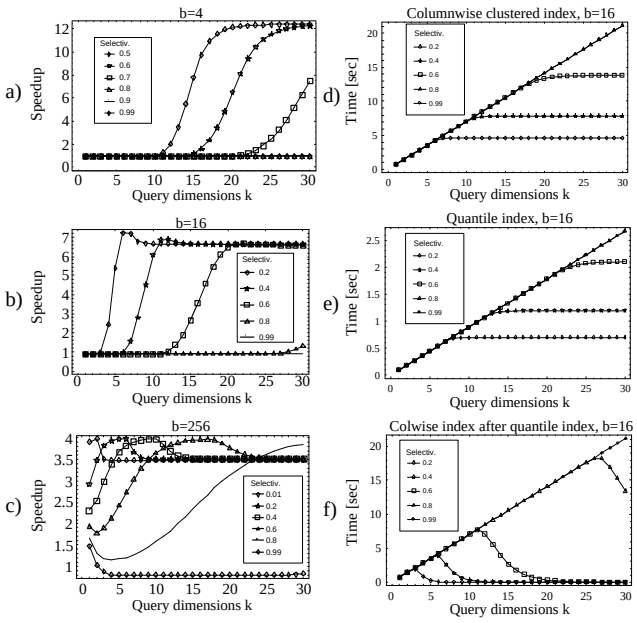

Figure 10: Exact search using quantile index

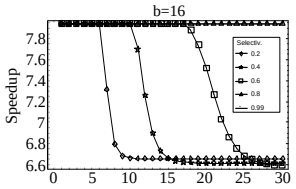

Figure 11: Probabilistic search using quantile index

up exact values of remaining candidates in a columnwise clustered index are shown in Fig. 10 f). Speedup rates are obtained by relating the time spent in d) to the time spent in e) + f). Parameters again were $N = 10^6, pagesize = 4096, as = 4$ (float attributes).

6.3.2 Analysis of probabilistic search

Fig. 10 d) and e) also compare probabilistic search with a confidence threshold of 0% (the worst case in terms of I/O) against a plain columnwise clustered index. From those quantities the speedup rates depicted in Fig. 11 can be calculated. The maximum speedup we can hope for with $b = 16, as = 4$ is $\frac{as \times 8}{\lceil \log_2 b \rceil} = 8$. As expected, speedup is consistently close to the maximum, though not constant, due to the same reasons already stated in the analysis of the minmax index.

7 Conclusions

Let us now summarize in which areas of the parameter space the discussed indexes work well. The three studied parameters were: query dimensionality, result

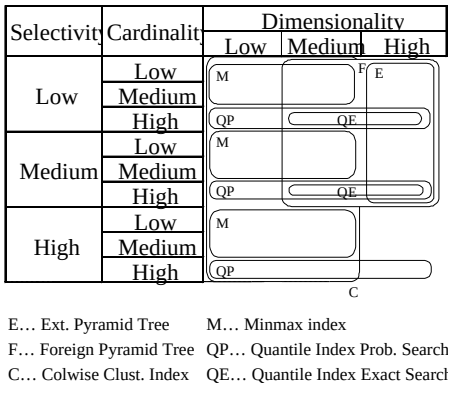

Figure 12: Parameter space covered

selectivity and cardinality of attribute domain. Figure 12 shows a simplified map, inaccurate by necessity. For details see the previous sections. The columnwise clustered index is applicable for many use cases. The minmax and quantile index support special use cases where they improve on the columnwise clustered index. The foreign pyramid tree enriches the applicability of the extended pyramid tree.

We studied physical database design for OLAP range queries. A useful DBMS should support the continous spectrum from low to high dimensional range queries with small to large selectivities operating on low to high cardinality attribute domains, a problem not solvable by one single index structure. A set of heuristics was collected: multidimensional indexes are good for queries on most of the indexed dimensions, clustered files are good for queries on few dimensions, lossy compression can be useful for pruning, as well as other heuristics. We then suggested judiciously combining these observations with the state of the art in multidimensional indexes, bitmap indexes, lossily-compressed indexes, and others, to provide a design that supports a broad variety of range queries. Our technique is a hybrid composed of a variant of the extended pyramid tree, columnwise clustering, minmax based lossless and bin based lossy compression. This combination offers advantages over individual indexes, as the advantages of each structure can be multiplied.

Our building blocks improve on their ancestors in several ways. Whereas the (extended) pyramid tree is limited to queries close to exact range queries, the foreign pyramid tree can effectively be used for exact and partial range queries without duplicating data objects. For low and medium cardinality attribute domains, the minmax index shows strong speedups over a plain columnwise clustered index. Next, the approximate index attacked the problems caused by high cardinality attribute domains. It is a framework modelling the constraints under which any lossily compressed index operates when obtaining both probabilistic and exact

answers, regardless of the underlying implementation. Last, a concrete implementation of an approximate index was proposed, the quantile index. For probabilistic search it shows strong speedup rates over a column-wise clustered index. For exact seach it can often but not always contribute moderate to large speedups. All index data structures are particularly designed to a) complement each other and b) to be combinable in a filter pipeline.

Ongoing work focusses on combining the individual cost models to establish global cost models for global execution plans. Further papers will report on the combined power of this approach. We are currently implementing an OLAP warehouse for the high energy physics experiment NA48 at CERN. Detailed experimental and production results will be published as soon as available.

References

[Bay97] R. Bayer. The universal B-tree for multidimensional indexing: General concepts. *Lecture Notes in Computer Science*, 1274, 1997. Available at mistral.informatik.tu-muenchen.de/publications/index.html.

[BDF+97] D. Barbar, W. DuMouchel, Ch. Faloutsos, P. J. Haas, J. M. Hellerstein, Y. E. Ioannidis, H. V. Jagadish, T. Johnson, R. T. Ng, V. Poosala, K. A. Ross, and K. C. Sevcik. The new jersey data reduction workshop report. *IEEE Data Engineering Bulletin*, 20:3–45, 1997. Available at ftp.research.microsoft.com/pub/debull/dec97-letfinal.ps.

[BK98] S. Berchtold and D. A. Keim. Tutorial: High-dimensional index structures - database support for next decade's applications. In *ACM SIGMOD Int. Conf. on Management of Data*, 1998. Available at www.research.att.com/∼berchtol.

[BKB98] S. Berchtold, H.-P. Kriegel, and C. Böhm. The Pyramid-tree: Breaking the curse of dimensionality. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, 1998. Available at www.research.att.com/∼berchtol.

[BKK96] S. Berchtold, D. A. Keim, and H.-P. Kriegel. The X-tree: An index structure for high-dimensional data. In *Proc. of the 22nd Int. Conf. on Very Large Data Bases*, 1996. Available at www.research.att.com/∼berchtol.

[CD97a] S. Chaudhuri and U. Dayal. An overview of data warehousing and OLAP technology. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 26(1), 1997. Available at www.research.microsoft.com/users/surajitc/.

[CD97b] Surajit Chaudhuri and Umeshwar Dayal. Data warehousing and OLAP for decision support (tutorial). In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, 1997. Available at www.acm.org/pubs/contents/Proc./mod/253260/index.html.

[Cha98] Surajit Chaudhuri. An overview of query optimization in relational systems. In *PODS '98. Proc. of the 17th ACM SIG-SIGMOD-SIGART Symposium on Principles of Database Systems*, Seattle, Washington, June 1998. ACM Press. Available at www.research.microsoft.com/users/surajitc/.

[CI98] Chee Yong Chan and Yannis E. Ioannidis. Bitmap index design and evaluation. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, 1998. Available at www.cs.wisc.edu/∼cychan/101.ps.

[CMS96] CMS. Compact muon solenoid computing technical proposal. December 1996. Available at cms-doc.cern.ch/ftp/CMG/CTP/index.html.

[Col96] Compass Collaboration. The compass proposal, addendum 1. May 1996.

[Com79] Douglas Comer. Ubiquitous B-tree. *ACM Computing Surveys*, 11(2):121–137, June 1979.

[Fre95] M. Freeston. A general solution of the n-dimensional B-tree problem. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 24(2), June 1995.

[GG96] Volker Gaede and Oliver Guenther. Multidimensional access methods. Technical Report TR-96-043, International Computer Science Institute, Berkeley, CA, October 1996. Available at www.wiwi.hu-berlin.de/∼gaede/vg.pub.html.

[Gup97] Vivek Gupta. An introduction to data warehousing. *System Services Corporation*, August 1997. Available at system-services.com/dwintro.htm.

[Hol97] Koen Holtman. Selective read. *Using an Object Database and Mass Storage System for Physics Analysis, CERN/LHCC*, 1997. Available at wwwinfo.cern.ch/pl/cernlib/ rd45/reports.htm.

[Hol98] Koen Holtman. Clustering and reclustering hep data in object databases. In *Proc. of CHEP '98*, 1998. Available at home.cern.ch/k/kholtman/www/.

[LJF94] King-Ip Lin, H. V. Jagadish, and Christos Faloutsos. The TV-tree: An index structure for high-dimensional data. *VLDB Journal*, 3(4), October 1994. Available at www.msci.memphis.edu/∼linki/index.html.

[Mal98] David Malon. Personal communication. 1998.

[MRL98] Gurmeet Singh Manku, Sridhar Rajagopalan, and Bruce G. Lindsay. Approximate medians and other quantiles in one pass and with limited memory. In *Proc. of the 1998 ACM SIGMOD Int. Conf. on Management of Data*, 1998. Available at www.cs.berkeley.edu/∼manku/papers/quantiles.ps.gz.

[OQ97] Patrick O'Neil and Dallan Quass. Improved query performance with variant indexes. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, 1997. Available at www.cs.umb.edu/∼poneil/publist.html.

[PS98] Bernd Panzer-Steindel. Central data recording for high data rate experiments at cern. In *CHEP Computing in High Energy Physics*, 1998.

[Sch94] Erich Schikuta. Parallel relational database algorithms revisited for range declustered data sets. In *Proc. of the 1994 ISPAN Int. Symposium on Parallel Architectures, Algorithms and Networks*, Kanazawa, JP, December 1994. Available at www.pri.univie.ac.at/∼schiki/es-research.html.

[Syb97] Sybase. Sybase iq indexes. In *Sybase IQ Administration Guide, Sybase IQ Release 11.2*, 1997.

[WB98] Ming-Chuan Wu and Alejandro P. Buchmann. Encoded bitmap indexing for data warehouses. In *Proc. of the 14th Int. Conf. on Data Engineering*, 1998. Available at www.informatik.tu-darmstadt.de/DVS1/staff/wu/wu.htm.

[WSB98] R. Weber, H.-J. Schek, and S. Blott. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *24th Int. Conf. on Very Large Databases*, 1998. Available at www-dbs.ethz.ch/∼weber.

[WY96] Kung-Lung Wu and Philip S. Yu. Range-based bitmap indexing for high cardinality attributes with skew. Technical report, IBM Watson Research Center, May 1996. Available at domino.watson.ibm.com/library/ CyberDig.nsf/Home/.